JITTA

JOURNAL OF INFORMATION TECHNOLOGY THEORY AND APPLICATION

ISSN: 1532-3416

Effort Estimation Factors for Corrective Software Maintenance Projects: A Qualitative Analysis of Estimation Criteria

Michael J. Lee
David Eccles School of Business
University of Utah, USA
mike@strategicdatainsights.com

Marcus A. Rothenberger
Lee Business School
University of Nevada Las Vegas, USA

Ken Peffers
Lee Business School
University of Nevada Las Vegas, USA

Abstract:

In this paper, we identify factors that impact software maintenance effort by exploring expert software maintenance estimators' knowledge about corrective maintenance projects. We use a qualitative approach to identify the issues important to these experts to derive their effort estimates. We find seventeen factors (rated and rank ordered by importance) that affect corrective maintenance effort and include constructs related to developers, code, defects, and environment. Several of these factors that have a comparably strong influence on corrective maintenance estimation are unique to corrective maintenance and are not generally observed in established software estimation models. The results enhance organizations' ability to effectively manage maintenance environments by focusing attention on the identified areas. For future research, these results represent an important step toward developing a comprehensive and accurate corrective maintenance effort estimation model.

Keywords: Maintenance Management, Maintenance Planning, Estimation Factors, Effort Estimation.

Jan vom Brocke acted as the Senior Editor for this paper.

1 Introduction

Software is expensive, and software's most significant expense over its lifecycle is related to maintenance (Banker & Slaughter, 2000; Mukhopadhyay, Vicinanza, & Prietula, 1992). This cost can be substantial, and an organization's ability to predict and control software maintenance effort is a critical part of their risk management strategy (Boehm & Papaccio, 1988). Maintaining software also takes time, and estimating the effort needed to do so is difficult. For a maintenance program to be considered successful, maintenance releases must be delivered regularly and predictably (Sneed & Brössler, 2003). Therefore, understanding which factors influence effort is vital to accomplishing these maintenance tasks, which we focus on identifying in this paper.

Not every maintenance intervention is worth making. Some defects are not worth fixing, and some adaptations are not cost-effective. Thus, one should estimate the effort associated with those interventions in advance in order to perform the analysis necessary to determine if interventions are appropriate. Unfortunately, success in software estimation generally, and in maintenance particularly, has been elusive in that it has been plagued with complex models that are characterized by high result variance and a lack of practical relevance (Menzies, Chen, Hihn, & Lum, 2006). Thus, in practice, managers often struggle with providing accurate estimates for maintenance activities. Even more fundamental, however, are the decisions that managers must make every day on how to best organize and equip a code maintenance team to ensure the highest levels of productivity. Without a thorough understanding of the factors that impact maintenance effort, a manager would make these decisions with little or no guidance. Providing a structure for these decisions can potentially improve the quality and delivery of a maintenance code. In this paper, we identify these factors, which can assist practice in the daily management of maintenance effort, and provide a foundation for future research related to maintenance model development.

Researchers have identified three types of maintenance interventions, each with its own objectives and processes (Bandi, Vaishnavi, & Turk, 2003): corrective, adaptive, and perfective maintenance. Corrective maintenance refers to modifying a system to ensure that it functions according to intended specifications. This is sometimes referred to as bug-fixing. Adaptive maintenance comprises modifications made to a system to alter that system in order to accommodate changing environments such as hardware, operating systems, or other environmental factors that can affect the system's functionality. Finally, perfective maintenance interventions are intended to meet changing user requirements to ensure that, as user needs change, the system will still meet their needs. Table 1 summarizes the three intervention types.

Research suggests that each of the three intervention types—corrective, adaptive, and perfective maintenance—should have its own estimation models (Fioravanti & Nesi, 2001) because each type requires a substantially different set of tasks and skills. Adaptive and perfective maintenance tasks can potentially benefit from standard software estimation models or models extended from standard models because their lifecycle process of design and implementation based on new requirements is similar to the lifecycle process of new development (De Lucia, Pompella, & Stefanucci, 2005). However, corrective maintenance is different in that it does not seek to implement any new requirements but rather repair the

Contribution:

This paper contributes to both research and practice by providing details about the decision making considerations of software maintenance experts that they use to make their estimates. Many of these factors have not been identified by previous research or models and, therefore, represent a unique contribution of this research. Corrective maintenance, or bug-fixing, is an understudied area in software estimation literature; however, the factors that impact estimation of development efforts to perform corrective maintenance activities are often not common to other software estimation influence factors and, therefore, require separate consideration. We use a qualitative approach to develop these factors through Web-based interaction with software maintenance experts, which is rigorous and follows best practice for this type of inquiry. Since this approach is not commonly applied in information systems research, it not only reveals insights into software maintenance estimation factors but also provides an informative example of how to pursue a Web-based qualitative inquiry. Since many of the factors revealed are environmental factors, this research also makes direct contributions to practice because it informs managers regarding the optimal organizational and environmental conditions to maximize the accuracy of estimates and promote efficiency in software maintenance teams.

application to ensure that existing requirements are implemented correctly. In corrective maintenance, therefore, much of the effort is shifted from design and coding to debugging and diagnosis. As a result, corrective maintenance is much more difficult to estimate because the predominant estimation techniques are largely inappropriate. For example, the maintainer may spend substantial time identifying the cause of a defect only to make a one-line change to the code. Metrics typically used in software estimation heavily weigh the costs of code change and, therefore, are of limited use for corrective maintenance. This is generally true regardless of the development methodology used. Agile development methodologies are becoming increasingly popular; however, agile iterations for deployed products that represent implementation of new requirements are best classified as either adaptive or perfective maintenance, depending on the nature of the new requirements. Even if corrective maintenance activities are part of an agile iteration, they must still be estimated as part of the iteration, and the same challenges related to the effort estimation of these corrections apply to the overall estimation of the iteration.

Table 1. Maintenance Intervention Types

Intervention type	Objective of Intervention
Corrective	Repair defects that cause the application behavior to deviate from stated specifications. **No new requirements.**
Adaptive	Adapt the application to a new environment, such as a new operating system or device firmware. Often requires implementing new nonfunctional requirements. An example is porting a Windows-based application to a Linux platform.
Perfective	Application modifications made to accommodate new or changing requirements. New functional and nonfunctional requirements are, by definition, required for this type of intervention. Agile methodologies, which implement new requirements iteratively, often heavily use this intervention type.

Notwithstanding the distinction between corrective and adaptive/perfective maintenance, little work has been published on developing models specifically for corrective maintenance. Most prior research takes a general approach to maintenance estimation (e.g., Mukhopadhyay et al., 1992; Smith, Hale, & Parrish, 2001). DeLucia et al.'s (2005) study is a notable exception: it is devoted specifically to corrective maintenance. The study explores existing corrective effort estimation model implementations at two companies, compares projections with actual values to identify variances, and develops new, more accurate models based on the existing factors identified in the existing models. While DeLucia et al. (2005) show improvements and reduced variances, they do not attempt to extend the existing model implementations by identifying and introducing new estimation factors, which creates an opportunity for this type of research.

To learn which factors expert estimators consider when making corrective maintenance project estimates, we use the collective causal mapping methodology (CCMM) (Scavarda, Bouzdine-Chameeva, Goldstein, Hays, & Hill, 2006), a qualitative approach based on causal mapping theory (Axelrod, 1976) and the Delphi method (Dalkey & Helmer, 1963) that can concisely capture which factors experts consider to arrive at their corrective maintenance effort estimates. This methodology incorporates best practices of qualitative investigation and is consistent with Miles and Huberman (1994) to ensure appropriate rigor for qualitative investigation.

2 Software Estimation

The literature features numerous software estimation models. The most established models are the software lifecycle management (SLIM) model (Putnam, 1978) and the constructive cost model (COCOMO) (Boehm, 1981). Over the years, these models have been updated to accommodate changes in technology and methodology. For example, COCOMO II (Boehm et al., 2000) revises and enhances Boehm's initial work. The movement to object-oriented development has also required changes to these early models to keep them relevant, and early authors have frequently revisit their work as technology has changed (Boehm & Valerdi, 2008). Much of the research in software estimation is based on this early work and has interesting augmentations that concentrate on certain aspects of software cost. As an example, In, Baik, Kim, Yang, and Boehm (2006) propose a quality-based estimation model called the quality-based software product line cost estimation model (qCOPLIMO), which is based on two COCOMO

suite models. In et al.'s model considers software quality costs in the context of existing COCOMO models and uses quality as a factor that affects cost.

Despite the wide availability and diversity of estimation models and studies (Jørgensen & Shepperd, 2007), observed variances between predicted and actual values remain high (Menzies et al., 2006), which provides support research that attempts to enhance these models or develop entirely new estimation methods to provide more accurate estimations.

3 Motivation and Theory

As we note in Section 2, corrective maintenance estimation, while somewhat related to software estimation, differs substantially from development estimation. Thus, while some extrapolations can be made from software estimation to the study of maintenance estimation, corrective maintenance requires its own research and models. Early research in maintenance focused on differentiating development and maintenance tasks. Kemerer and Slaughter (1999) have encouraged new research on software maintenance processes due to the important distinction between software maintenance and software evolution. They describe maintenance as the modification necessary to ensure that software met its original intent, while evolution is the modifications necessary to extend the reach of a system into new areas. This definition supports identifying corrective maintenance as a distinct process when compared with adaptive or perfective maintenance. Under this definition, only corrective maintenance would truly be classified as maintenance, while perfective and adaptive maintenance would be classified as evolution. This classification is logical when we consider the role of agile methodologies in current practice. While agile iterations may include adaptive or perfective tasks, they are generally considered "evolutionary" enhancements to the existing product and not maintenance by the strictest definition. Corrective maintenance, again, stands alone in this regard.

Corrective maintenance focuses on repairing defects rather than expanding a system's intended purpose. It differs from other maintenance intervention types in that traditional software estimation models are less applicable because of the extensive amount of time spent on identifying the defect and debugging it. Research suggests that such cognitive and managerial functions play an important role in the performance of software maintainers (Jørgensen, 1995); therefore, factors relating to these issues should be included in effort estimation of corrective maintenance. In fact, Nguyen, Boehm, and Danphitsanuphan (2011) report that more time is typically spent on task and code comprehension activities for corrective maintenance than for other maintenance types. Consequently, an opportunity exists for further research that explores the factors considered by expert estimators, exposing the distinct factors that are important for corrective maintenance effort estimation. Ultimately, this understanding may lead to improvements in corrective maintenance estimation. In this study, we capture the factors employed by corrective maintenance experts to arrive at their estimates, and we use that information to determine which factors might truly be of interest when promoting an environment that is optimal for corrective maintenance tasks.

The theoretical foundation for taking an expert judgement-based approach to this problem is strong. Hammond (1986) describes that cognition is on a continuum rather than being dichotomous, which implies that individual experts are usually not either right or wrong but that their opinions lie on a continuum of truth that, when aggregated, can provide a better approximation of reality. This provides support for an expert panel approach for determining the rationale for decision-making. While it is only one method, in the absence of available data for empirical analysis, it can at least provide a foundation for further testing and evaluation. It is on this theoretical basis that we can move forward with an expert judgement-based approach for identifying factors in the decision making process related to maintenance effort estimation.

4 Research Methodology

For this paper, we followed the causal mapping methodology, a qualitative approach used to identify the criteria that individuals employ to accomplish a goal or reach a decision. The foundations for this approach, pioneered by Axelrod (1976), state that, to comprehend the thought process of experts, one must understand the causal links that they use to reach their decisions. In fact, cognitive causal mapping techniques are a way of exposing the human factors that often underlie the technical concerns in information systems (Siau & Tan, 2008).

In this paper, we employ the collective causal mapping methodology (CCMM) (Scavarda et al., 2006), which takes a virtual approach to causal mapping by using Web-based interactions with participants as

opposed to traditional interviews. Through Web-based interviews and interactions with software maintenance experts, we identify and rank-order a set of factors that contribute to corrective maintenance effort. CCMM provides a complete set of guidelines that define the study progression, including how to construct the Web-based interview instruments, techniques for coding the resulting unstructured data, and organizing this data into a weighted causal map. CCMM's Web-based interaction paradigm of the CMM has certain advantages over a traditional interview-based technique. It allows the researcher to work with a larger, more geographically dispersed pool of experts. The experts can remain completely anonymous, and, because all communication is handled electronically, there are no interactions directly among the respondents. This eliminates the possibility of groupthink, which can negatively impact the exchange of ideas in direct group interaction. As a result, the CCMM adheres to the general requirements for rigor when gathering data using Delphi-based techniques such as geographical diversity, participant anonymity, and the ability to provide precise and consistent instructions (Paré, Cameron, Poba-Nzaou, & Templier, 2013).

4.1 Participants

We sent invitations to professionals who we knew had expertise in software maintenance estimation, specifically in object-oriented programming. We drew these participants from several different geographical areas; specifically, the Southwestern, Southern, and Midwestern United States and Western Canada. They also represented diverse industries, including financial services, insurance, government, nonprofit, entertainment, manufacturing, and gaming. The participants worked in different roles, including quality assurance specialists, developers, project managers, development managers, and technical executives. One of the researchers' access to a substantial network of professionals throughout the US and Canada was instrumental in this effort to identify potential participants.

We selected participants purposefully rather than randomly. That is, we specifically selected individuals that would be able to provide the most substantial contribution to our understanding of corrective maintenance estimation while covering the domain of knowledge. This included ensuring that the recruited experts covered a wide range of backgrounds, roles, practices, and geographies. This selection strategy is not only viable but also necessary in qualitative research (Eisenhardt, 1989; Miles & Huberman, 1994; Seawright & Gerring, 2008). This approach is also consistent with the CCMM, which requires nonrandom participant selection to ensure that the participants' skills and abilities cover the subject domain.

4.2 Data Collection

To identify the factors that the participants believed impact maintenance effort and, therefore, their estimate of the effort to complete the maintenance task, we set up a website prompting participants to provide their insights on corrective maintenance estimation factors. The website presented a brief explanation on how to provide the requested information and then presented the participants with the task of identifying conditions that lead to outcomes related to corrective software maintenance effort. Participants were able to enter as many causal relationships as they found to be relevant to assessing the effort required to complete a corrective maintenance task. We expected direct causal relationships between the effort estimation factors and software maintenance effort (e.g., factor 1 causes high maintenance effort, factor 2 causes high maintenance effort, etc.); however, as this research is exploratory, we did not want to restrict the respondents into simple cause-effect patterns to account for the possibility that causal chains may exist. Thus, we used the CCMM's capability to ask the participant to enter their responses in a structured format that followed a pattern of "A causes B", where A is a condition and B is an outcome that the respondents filled in using free-form text. This format not only allowed participants to provide data as direct factors (e.g., factor 1 causes high maintenance effort) but also gave them the option to provide data in causal chains (e.g., factor 1 causes factor 2, factor 2 causes high maintenance effort). Nevertheless, few participants took advantage of the option of providing such causal chains. Most of the responses indicated a direct causal relationship of a factor to maintenance effort without providing intermediary factors. Those that did provide factor chains included no more than one intermediary factor; these responses communicated additional detail by explicitly naming an additional step in a causal chain, however, the nature of the relationship was consistently the same as that provided by other respondents who communicated the same relationship without intermediary factors. Because of the direct causal nature of these responses, we were able to eliminate much of the complexity from the model by collapsing it to a set of factors that directly impacted maintenance effort. The identified factors provided the most parsimonious interpretation of the data and confirmed our expectation of a direct causal relationship between effort estimation factors and software maintenance effort.

We conducted an initial pilot study to evaluate the data collection approach. Using the feedback and the results of this pilot study, we adjusted the wording of the instrument to ensure that the participants provided relevant data in the correct format. Based on the results of this pilot study, we sent invitations to 41 potential participants, which generated a total of 27 responses. Three respondents in the study appeared to have misunderstood the nature of the inquiry and did not provide any useful information. Therefore, we removed them from the data set, resulting in 24 usable responses. As we show in Section 4.4, our analysis was saturated with this initial set of participants, so we did not need to recruit additional participants for this part of the study.

The respondents' ages ranged from 27 to 55. Their reported maintenance experience varied from six to 31 years. Their experience in object-oriented programming maintenance ranged from four to 16 years. In accordance with the CCMM, we also asked the participants to self-report their level of proficiency in software maintenance on a 7-point Likert scale (1 = "not proficient", 4 = "moderately proficient", and 7 = "extremely proficient"); they reported proficiency ranging from 4-7. All participants met the inclusion criterion of having substantial practice in software maintenance of object-oriented systems.

4.3 Coding

The first and second authors, each having extensive experience with software development, maintenance, and object-orientation, independently coded the responses into categories. Because this was an exploratory study, and to be consistent with the CCMM, we defined no categories in advance. Rather, we defined the categories as suggested by the data (open coding). This approach to coding and classification is common and generally accepted in qualitative/grounded theory methodologies. Indeed, Miles and Huberman (1994) and Strauss and Corbin (1990) support this approach. We expected each respondent to contribute only a subset of all categories to the final results because each participant may have had a unique experience that did not necessarily encompass all aspects of maintenance effort estimation. However, five participants stood out in that they had provided only one category each (compared to an average of 4.4 categories contributed by the other participants), which suggested that they were narrowly focused on one aspect of their experience and were not considering the bigger picture. This is consistent with Nelson, Nadkarmi, Narayanan, and Ghods' (2000) concept of the revealed causal map, which indicates that a person, for various reasons, may not fully disclose their entire experience. Nevertheless, in each of these five instances, the other respondents also provided the single factors named, and so we concluded that these five responses were valid contributions in spite of their rather narrow focus. Thus, we did not exclude these five responses from the analysis; this decision did not affect the results because of the support of these respondents' factors by multiple participants.

We made the deliberate decision to enter all categories identified by this process into the next step of the analysis even if they were named only by one respondent; this happened in two instances. We reasoned that that the existence of a subsequent ranking phase in our analysis allowed us to be inclusive at this stage, accounting for the possibility that even a factor named by only one participant may be important. We then determined the final importance of a factor in the ranking. Other participants may have failed to name a relevant factor, possibly because of the revealed causal map concept discussed earlier or because of a lack of firsthand experience with the respective issue. Yet, when prompted to rank such a factor, they may recognize its importance. In fact, one of the factors identified by only one respondent was ranked relatively high (6 out of 17), while another such factor ranked low (16 out of 17).

We (first and second author) coded over 88 percent of the respondents' observations identically. We resolved the remaining 12 percent after one round of discussion, resulting in 100 percent agreement. All coding was done incrementally, with the researchers always reviewing cases in the same order. When discussion resulted in a modification to the coding categories or definitions, we restarted the process and considered each of the cases again, in the same order, to ensure that all cases were compliant with the new categories and definitions. This is an integral part of the open-coding approach because any new information may require the coders to reevaluate existing data from a different perspective. Strauss and Corbin (1990) encourage this approach and it adds rigor to the coding process. In a subsequent step, the third author audited the results by confirming the codes. This researcher independently assigned participant observations to the defined categories. Using a third cover to validate coding is the generally accepted method for validating the initial coding results as per Miles and Huberman (1994). This step revealed that four observations were stated ambiguously (fitting in either of two existing categories); thus, we excluded the observations from the analysis. These exclusions did not affect the results (affected categories were supported by multiple other observations). One inconsistency led us to reword a category

definition for clarity. We resolved the five remaining inconsistencies, representing only 4 percent of the observations that were entered in the analysis, in one iteration of clarification with the audit coder, who agreed with the initial coding on those observations. We present and discuss these final audited factors in Section 4.4. Table 2 provides their definitions.

Table 2. Definitions of Causal Factors

Category	Factor (ID)	Factor definition
Developer-related factors	Low developer familiarity with the product (A)	The developer has a low level of familiarity with the code, code structure, or business domain of the product.
	Low developer familiarity with the technology (B)	The developer has a low level of familiarity with the programming language, platform, or associated technologies used in the product.
	Low developer experience in maintenance (C)	The developer is less skilled or experienced in designing, developing, or debugging.
Code-related factors	High code complexity (D)	The code being maintained is structurally complex, uses complex patterns or technologies, or is large in size.
	Low clarity of code structure (E)	The affected code has been designed or implemented in a way that limits its structural clarity.
	High level of code/system dependencies (F)	The code being maintained has substantial dependencies to other systems, components, or code.
	High version/deployment complexity (G)	The code being maintained is present in many supported/ deployed versions of the product.
	High level of code volatility (H)	The code being maintained is experiencing a high level of churn/change not related to the defect.
	Low availability of formal design documentation and code comments (I)	There is only limited availability of design documentation, including models, diagrams, use cases, etc., or they are not available, or the code is not well commented.
Defect-related factors	Low clarity or availability of defect documentation (J)	Documentation of the defect behavior is low; availability of logs and/or access to stakeholders for clarification is low.
	Low defect reproducibility (K)	The defect is not easily reproducible in a maintenance environment.
	Low code coverage of unit tests (L)	At the beginning of the maintenance project, few unit tests are available to test, validate, or regress behavior.
Environment-related factors	High regulatory impact (M)	The code being maintained covers a feature or functionality that has high legal or regulatory impact on the business.
	Low perception of defect criticality by management (N)	Management views the defect's correction to be of low criticality or low priority.
	High level of task switching (O)	The developer or team has responsibilities not related to fixing the defect and must frequently switch between assignments.
	Low level of team cohesion (P)	The team does not collaborate or coordinate their efforts well.
	Low availability of required tools (Q)	There is little access to tools such as debuggers, libraries, compilers, etc.

4.4 Response Saturation

One concern frequently associated with qualitative methods is whether sufficient data has been collected to ensure that the research has captured the maximum amount of data that is practically possible to collect. Eisenhardt (1989) refers to this point as theoretical saturation. CCMM provides a method for estimating the level of saturation of causal relationships obtained from additional responses by using a nonlinear least squares curve fit model that predicts the number of relationships obtained from *n* respondents.

$$R(n) = \alpha(1-e^{-\beta n}) \quad (1)$$

In our research, this regression, with an α estimate of 16.240 and a β estimate of 0.190, demonstrates a good fit to the data with an R^2 of 94.5 percent. Using this model, the addition of a 25 participant into the analysis would generate an estimate of a marginal increase of .01 new factors for the next respondent, which represented a marginal percentage increase of .09 percent. Thus, we concluded that additional respondents would be unlikely to expand the model and that we reached theoretical saturation. Although we coded the data incrementally as we received it, we did not run a saturation analysis until we had gathered responses from 25 participants—all usable responses from our initial recruitment effort. Since the regression estimated an extremely low potential marginal gain from additional participants, we discontinued recruiting new participants at this point.

The data analysis resulted in our identifying seventeen causal relationships that impact effort in corrective maintenance. These relationships represent a concise interpretation of the data through both the first two researchers' initial coding and the third researcher's audit coding; thus, no further clustering of the data was likely. The CCMM provides for an optional cluster step that allows one to further collapse codes. However, because of the results' concise nature, we did not need to further consolidate the codes. We included a factor in the results as long as at least one respondent cited the factor as causal to software maintenance effort. We set the inclusion threshold deliberately low because the CCMM provides a process for determining the strength of each relationship by participants' vote. Thus, an inclusive approach at the initial step that identifies the factors to be voted on was appropriate. Table 3 lists the confirming observations from the participants for each factor.

4.5 Factor Ranking

Understanding the relative strength is critical in interpreting the results so that any estimation or management models derived from the results can focus primarily on the higher rated factors. Therefore, once we identified and defined the factors, we next rank ordered the factors based on input from our pool of experts. We created a new survey page that presented each of the seventeen factors in a different random order, along with their definitions, to the participants. We designed the webpage so that the seventeen factors would be randomized for each visit. Therefore, even if the same participants were to return to the survey again to modify their responses, the factors would be presented in a different order from the ones they previously observed. This randomization prevented any possible positional bias from presenting itself in the results. The survey asked the participants to rate how strongly each factor affected maintenance effort on a 7-point Likert scale (1 = "extremely weak", 4 = "moderate", and 7 = "extremely strong").

We sent invitations to the original pool of 41 experts and an additional nine experts in an effort to maximize the number of ranking responses received. CCMM supports and encourages one to use original participants and additional participants at this stage to provide a larger sample (Scavarda et al., 2006). These 50 invitations generated a total of 31 responses. The respondents reported maintenance experience from one to 30 years (one to 16 years of object-oriented technology). Their self-reported proficiency ranged from 4 to 7 on the 7-point Likert scale previously described.

Defined in the CCMM is a process for scoring the relationships under study, which, in this research, are the factors previously identified as having an impact on maintenance effort. The scoring process considers that factor ratings obtained from highly experienced respondents may be more valuable than those obtained from less-experienced participants and, thus, that they should be weighted more strongly in a cumulative assessment of factor strength. According to the CCMM approach, we weighted the rating scores based on the experience and reported proficiency of the respondent. The cumulative strength of each factor is reported as a normalized measure between 0 and 1, where 0 corresponds to the value of 1

Table 3. Factors and Participant Responses

	Factor ID[1]																
Participant number	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**
1				X													
2	X		X		X					X						X	
3	X		X	X						X	X				X		X
4					X												
5	X					X					X						
6	X		X											X	X		
7	X			X							X						
8	X				X			X				X					
9	X	X				X				X		X	X		X		
10	X					X									X		X
11						X											
12				X	X												
13	X	X			X							X					
14	X	X	X	X													
15	X			X						X							
16	X																
17						X	X				X		X				
18	X					X											
19	X			X		X						X					
20					X												
21	X				X					X							
22	X		X				X		X	X	X	X					
23	X	X			X	X					X				X		X
24	X	X	X											X	X	X	X

[1] Factor ID values in column headers are provided in the Factor column of Table 2.

on the Likert scale, or "extremely weak", and 1 corresponds to the value of 7 on the Likert scale, or "extremely strong". We calculated the normalized strength of each factor (w_{jk}) by using an expertise factor (e_i) of each respondent as the weight for the rating feedback that each respondent provided for each factor (x_{ijk}) using the following formula:

$$w_{jk} = \left(\sum_{i \in R_{jk}} e_i x_{ijk} / x_{\max} \right) / \sum_{i \in R_{jk}} e_i \qquad (2)$$

,

where R_{jk} is the set of respondents that rated the relationship (j,k) of the factor to maintenance effort and $x_{\max}$ is the maximum rating for any factor, which in this study is 7. The expertise factor (e_i) is a function of each respondent's years of experience and the respondent's self-reported proficiency level. The measure incorporates the concept of diminishing margins for experience and increasing margins for proficiency. In other words, the number of years of experience of a respondent has diminishing returns as the number increases, while the value of self-reported proficiency increases as the number increases. To account for

these trends, Scavarda et al. (2006) incorporates the exponential factors α and β into their model and calibrate them with α = .5, providing diminishing returns for experience, and β = 2, providing increasing returns for self-reported proficiency. Since we use the same experience measures in our study, we decided to use the same calibration as Scavarda et al. This expertise factor formula incorporates these measures as follows:

$$e_i = (y_i/y_{\max})^{\alpha} \, (s_i/s_{\max})^{\beta} \quad , \qquad (3)$$

where y_i is the years of experience reported by the respondent, $y_{max}$is the maximum years of experience reported by any respondent (30 in this study), s_i is the self-reported proficiency of a respondent, and s_{max} is the maximum self-reported proficiency of any respondent (7 in this study).

5 Results

The first phase of the study obtained a set of estimation factors based on the cumulative experience of maintenance experts. The second phase of the study obtained factor ratings from the participants that lead to a rank-order of estimation factors based on their relative impact on maintenance effort. The results of both phases of the study are discussed in this section.

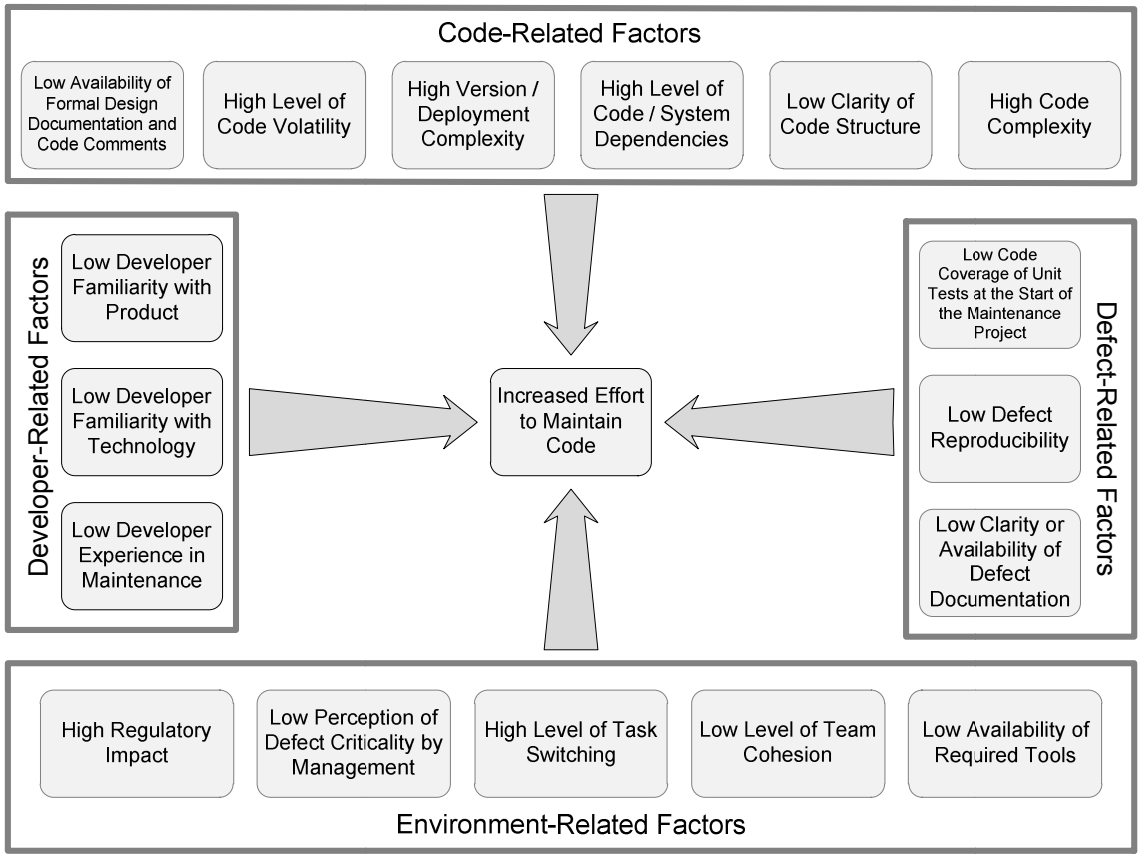

Figure 1. Identified Causal Factors Organized into Four General Categories)

5.1 Factors

The initial phase of this research produced a total of seventeen factors that impact corrective maintenance effort. Figure 1 illustrates these factors and groups them into categories. To define these categories, the first and second author independently arranged the factors into groups based on the general characteristics of each factor. The categories we produced were consistent with each other, and, therefore, we adopted the categories as a classification. Table 2 presents the definition of each node and its relationship to maintenance effort as reported by the experts who provided input to this study.

Table 4. Corrective Effort Estimation Factors, Rank Ordered by Weighted Standardized Assessment of Impact on Maintenance Effort, Within Category Classification

Rank	Weighted standardized response	Developer-related factors	Code-related factors	Defect-related factors	Environment-related factors
1	0.8027		High code complexity		
2	0.7812		Low clarity of code structures		
3	0.7539	Low developer experience in maintenance			
4	0.7537			Low defect reproducibility	
5	0.7080		High level of code/ system dependencies		
6	0.7069		High level of code volatility		
7	0.7020	Low developer familiarity with product			
8	0.6878				Low availability of required tools
9	0.6732	Low developer familiarity with technology			
10	0.6729			Low clarity or availability of defect documentation	
11	0.6721				High level of task switching
12	0.6508		High version/ deployment complexity		
13	0.6231				Low perception of defect criticality by management
14	0.6072			Low code coverage of unit tests	
15	0.5985				High regulatory impact
16	0.5945		Low availability of formal design documentation and code comments		
17	0.5838				Low level of team cohesion

5.2 Ranking

The second phase of the study provided a rank-ordered list of these factors based on participants' assessment of the impact of each factor on maintenance effort. The rank-ordered list (Table 4) reveals some interesting results with regard to the relationships that emerged at both the top and the bottom of the list. Since the list comprises factors that we obtained in the first phase of this study because of their relevance to software maintenance effort estimation, it is not surprising that the weighted standardized responses all indicate some degree of importance. In fact, it confirms the validity of the results obtained in the first phase. For example, the lowest factors scored .58 on a normalized scale from 0 to 1, where 0 represents the weakest estimate of factor impact and 1 represents the strongest. That means that even

the lowest ranked factor corresponds to a weighted cumulative strength rating between “moderate” and “strong” or between 4 and 5 on the 7-point Likert scale. For this reason, it would be inappropriate to dismiss any factor as irrelevant because even the lower-ranked factors are still considered at least moderately important to effort estimation.

6 Discussion

When we compare the estimation factors with the seventeen COCOMO II post-architecture effort multipliers (EM) (Boehm et al., 2000), we see both similarities and differences. Boehm et al. (2000) organize the EM set into four categories: product, platform, personnel, and project. Some of these drivers appear prominently in both the COCOMO II EM list and in our seventeen effort estimation factors. These include the complexity driver, the capability drivers, and the experience drivers. However, we find that, since we specifically targeted corrective maintenance activities, we see numerous factors in our study that are not present in Boehm et al. (2000) or in other standard software estimation models.

Most prominent in the list is the defect reproducibility factor. This is specifically related to corrective maintenance and is absent from established estimation models; however, it is critical to corrective maintenance estimation. The important issue with regard to this factor is its relative importance to other factors. Using the COCOMO II EM list to illustrate this point, only one of the factors in the COCOMO II experience drivers list (developer experience) was considered more important than reproducibility; however, this factor is absent from the model. Therefore, the contribution of this finding goes farther than simply identifying reproducibility as a factor; rather, it exposes the relative importance of the factor when compared to other factors in the established model. To extend this further, the developer experience factor, as the panel reported, was most relevant when related to experience with corrective maintenance specifically and not development experience overall. This essentially means that prior models do not capture some of the factors that the expert panel considers most important in providing an estimate.

The literature supports the importance of this finding. A description of the actions necessary to reproduce a defect is one of the issues that maintenance developers consider to be the most important in order to do their jobs, yet it is often challenging to provide (Zimmerman et al., 2010). If a defect is consistently reproducible, it is much easier to debug and isolate the offending code. If the error documentation does not provide these steps to reproduce the defect, then the developer must add time to the schedule to determine these steps.

Some of the factors categorized as developer-related are commonly used in both standard software estimation models and maintenance models (Boehm et al., 2000; Putnam, 1978). Issues related to some of these have also been discussed in the maintenance literature. For example, developer familiarity with the product and the technology has an obvious impact on the time required to complete a corrective maintenance task since these factors can potentially reduce the duration of cognitive activities such as task comprehension and defect isolation. This conclusion is consistent with Chua, Purao, and Storey (2006) who demonstrate that, by representing code structures in a more intuitive format, the maintenance task of code comprehension can be shortened and maintenance performance can be improved. Team cohesion is also supported in by the maintenance literature; Zhang, Stafford, Dahliwal, Gillenson, and Moieller (2014) address this issue in the context of the dynamics between developers and testers and find that many of the root causes for low team cohesion can be alleviated, potentially leading to a better performance of maintenance teams. Thus, this finding has direct implications for practice. Other factors are specific to corrective maintenance activities and are not generally found in established software estimation models, including code volatility, clarity of defect reports, and version complexity (multiple version management).

Another defect-related factor, “Low code coverage of unit tests”, appeared quite low in the rank order at position 14. As previously stated, we should not interpret this ranking to mean that the factor is not important. All the factors identified were scored with at least “moderate” importance by the panel, so this factor cannot be dismissed as irrelevant. The finding is still interesting, however, because there is ample practitioner literature advocating the use of unit testing, although most of it relates to unit testing in the context of test-driven development (TDD) and its impact on software quality (e.g., Crispin, 2006; Janzen & Saiedian, 2008). The practitioner literature rarely discusses unit testing in the context of software maintenance. Nevertheless, the prevailing perspective in the industry is that the purpose of unit testing is primarily for validating and regressing granular system functionality (Runeson, 2006). A possible explanation may be that, while unit testing may help developers produce better quality code, maintenance

developers would use unit testing primarily for regression tests of existing functionality that might be impacted by the corrective maintenance interventions needed to address a targeted defect. Consequently, an organization's unit testing program might reduce the total number of defects in a system during development, but, based on the results of our study, its use during maintenance has a lesser causal impact on the effort required to correct code defects that are discovered after release.

7 Conclusion

In this paper, we identify several estimation factors that are unique to corrective maintenance effort estimation and are not part of existing software development estimation measures. In addition, we identify certain factors known from the general software estimation literature that are applicable to corrective software maintenance. The results further provide a relative ranking of these factors. These findings have implications on both research and practice.

7.1 Contributions to Research

Many of the corrective maintenance effort estimation factors identified in this research differ from factors used in existing software development estimation models, such as the COCOMO II post-architecture effort multipliers. This underlines that the estimation of software maintenance effort is different from the more widely researched software development estimation. Thus, our results represent new and more applicable information for corrective maintenance estimation.

These findings provide an understanding of the relevant factors for corrective maintenance effort estimation and a relative ranking of these factors, which is an important foundation for future research to create an estimation model. Several studies suggest that multiple linear regression (MLR) provides the best vehicle for building estimation models (Fioravanti & Nesi, 2001; Jørgensen, 1995); however, the most critical challenge in creating such a model is in identifying appropriate measures for each of the estimation factors. While research may be able to draw from prior work to operationalize some of the effort estimation factors, other factors may be more difficult to measure because they have no established metrics. We have planned future work to address these issues to move forward toward creating a software maintenance effort estimation model.

We also believe that, in the context of Gregor (2006), this work makes a theoretical contribution to our understanding of the software maintenance estimation process. Gregor's "theory for explaining" category states that understanding how and why phenomena occur is an important contribution. Since we evaluate the effort estimation process in this paper, we feel that the rank-order list of factors produced by our process provide important understanding and explanatory value related to how these estimate are determined, especially since many of these factors are specific to maintenance issues and have not been revealed by previous research.

The results of this study prompt other interesting lines of future inquiry as well. For example, the software industry emphasizes unit testing. One of the reasons frequently cited for the necessity of unit tests is to simplify code maintenance. As we discuss in Section 6, the presence of unit test coverage in the code base does have a normalized score in the moderate range; however, it is one of the lower ranked factors, coming in at 14 of 17. While this does not suggest that unit tests are not valuable maintenance tools, it does certainly indicate that expert estimators think that many other factors impact maintenance effort more strongly than unit test code coverage. Additional research related to identifying the comparative value of unit testing for software activities, such as requirements management, maintenance, and development tasks, would certainly be valuable given that these results diverge from conventional wisdom.

7.2 Contributions to Practice

With this paper, we make two contributions to practice. First, with a better understanding of the factors that experts consider causal to corrective maintenance effort, managers can focus on identifying and leveraging metrics on those factors to provide better estimates, which requires organizations to understand and apply the metrics discussed previously. Second, managers can use their understanding of these factors to better manage the development environment to support more-efficient maintenance cycles. For example, understanding that expert estimators consider code complexity, system dependencies, and clarity of code structure to have a strong causal relationship to maintenance effort,

organizational resources can be concentrated in these areas. Beyond that, the soft factors impacting maintenance performance that we identify in this study may provide additional opportunity for managers to optimize their software maintenance environment. Being aware of soft factors, such as task switching, perceived defect criticality, and developer-familiarity/experience requirements, provides a manager with an opportunity to realign the maintenance teams or to alter the environment to promote productivity.

7.3 Limitations

To acquire participants with the experience and knowledge characteristics that we needed, we used a purposeful sampling technique in this study. Although this approach aims to maximize participants' diversity to capture all meaningful factors with regard to corrective maintenance effort, the possibility remains that the pool of participants was not diverse enough and that some factors may remain undiscovered. We addressed this potential concern by selecting participants from a wide range of industries, positions, levels of experience, and geographic classifications. This is the standard mitigation technique for this issue that qualitative research demands (Miles & Huberman, 1994). Although these results may not be generalizable to the entire population of software maintenance professionals, we have followed best practices in case selection to ensure that these results are as generalizable as possible. Nonetheless, there may be other contexts for software maintenance in which factors may play a role in software maintenance effort that we do not reveal here.

References

Axelrod, R. (1976). *Structure of decision: The cognitive maps of political elites*. Princeton, NJ: Princeton University Press.

Bandi, R., Vaishnavi, V., & Turk, D. (2003). Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Transactions on Software Engineering*, *29*(1), 77-87.

Banker, R., & Slaughter, S. (2000). The moderating effects of structure on volatility and complexity in software enhancement. *Information Systems Research*, *11*(3), 219-240.

Boehm, B. (1981). *Software engineering economics*. Englewood Cliffs, NJ: Prentice-Hall.

Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R, D., Reifer, D. J. & Steece, B. (2000). *Software cost estimation with COCOMO II*. Englewood Cliffs, NJ: Prentice-Hall.

Boehm, B., & Papaccio, P. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, *14*(10), 1462-1477.

Boehm, B., & Valerdi, R. (2008). Achievements and challenges in COCOMO-based software resource estimation. *IEEE Software*, *25*(5), 74-83.

Chua, C. E. H., Purao, S., & Storey, V. (2006). Developing maintainable software: The READABLE approach. *Decision Support Systems*, *42*, 469-491.

Crispin, L. (2006). Driving software quality: How test-driven development impacts software quality. *IEEE Software*, *23*(6), 70-71.

Dalkey, N., & Helmer, O. (1963). An experimental application of the delphi method to the use of experts. *Management Science*, *9*(3), 458-467.

De Lucia, A., Pompella, E., & Stefanucci, S. (2005). Assessing effort estimation models for corrective maintenance through empirical studies. *Information and Software Technology*, *47*(1), 3-15.

Eisenhardt, K. (1989). Building theories from case study research. *Academy of Management Review*, *14*(4), 532-550.

Fioravanti, F., & Nesi, P. (2001). Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. *IEEE Transactions on Software Engineering*, *27*(12), 1062-1084.

Gregor, S. (2006). The nature of theory in information systems. *MIS Quarterly*, *30*(3), 611-642.

Hammond, K. P. (1986). *A theoretically based review of theory and research in judgement and decision making*. Boulder: University of Colorado.

In, H. P., Baik, J. B., Kim, S. C., Yang, Y. D., & Boehm, B. . (2006). A quality-based cost estimation model for the product line life cycle. *Communications of the ACM*, *49*(12), 85-88.

Janzen, D., & Saiedian, H. (2008). Does test-driven development really improve software design quality? *IEEE Software*, *25*(2), 77-88.

Jørgensen, M. (1995). Experience with the accuracy of software maintenance task effort prediction models. *IEEE Transactions on Software Engineering*, *21*(8), 674-681.

Jørgensen, M., & Shepperd, M. (2007). A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, *33*(1), 33-53.

Kemerer, C., & Slaughter, S. (1999). An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, *25*(4), 493-509.

Menzies, T., Chen, Z., Hihn, J., & Lum, K. (2006). Selecting best practices for effort estimation. *IEEE Transactions on Software Engineering*, *32*(11), 883-895.

Miles, M. B., & Huberman, A. M. (1994). Qualitative data analysis: An expanded sourcebook (2nd ed.). Beverly Hills, CA: Sage.

Mukhopadhyay, T., Vicinanza, S., & Prietula, M. (1992). Examining the feasibility of a case-based reasoning model for software effort estimation. *MIS Quarterly*, *16*(2), 155-171.

Nelson, K., Nadkarmi, S., Narayanan, V., & Ghods, M. (2000). Understanding software operations support expertise: A revealed causal mapping approach. *MIS Quarterly*, *24*(4), 475-507.

Nguyen, V., Boehm, B., & Danphitsanuphan, P. (2011). A controlled experiment in assessing and estimating software maintenance tasks. *Information and Software Technology*, *53*(6), 682-691.

Paré, G., Cameron, A., Poba-Nzaou, P., & Templier, M. (2013). A systematic assessment of rigor in information systems ranking-type Delphi studies. *Information & Management*, *50*, 207-217.

Putnam, L. (1978). A general empirical solution to the macro software sizing and estimating problem. *IEEE Transactions on Software Engineering*, *4*(4), 345-361.

Runeson, P. (2006). A survey of unit testing practices. *IEEE Software*, *23*(4), 22-29.

Scavarda, A., Bouzdine-Chameeva, T., Goldstein, S., Hays, J., & Hill, A. (2006). A methodology for constructing collective causal maps. *Decision Sciences*, *37*(2), 263-283.

Seawright, J., & Gerring, J. (2008). Case selection techniques in case study research. *Political Research Quarterly*, *61*(2), 294-308.

Siau, K., & Tan, X. (2008). Use of cognitive mapping techniques in information systems. *Journal of Computer Information Systems*, *48*, 49-57.

Smith, R., Hale, J., & Parrish, A. (2001). An empirical study using task assignment patterns to improve the accuracy of software effort estimation. *IEEE Transactions on Software Engineering*, *27*(3), 264-271.

Sneed, H., & Brössler, P. (2003). Critical success factors in software maintenance. In *Proceedings of the International Conference on Software Maintenance* (pp. 190-198). Vienna, Austria: IEEE Computer Society.

Strauss, A., & Corbin, J. (1990). *Basics of qualitative research: Grounded theory procedures and techniques*. Newbury Park, CA: Sage.

Zhang, X., Stafford, T. F., Dahliwal, J. S., Gillenson, M. L., & Moieller, G. (2014). Sources of conflict between developers and testers in software development. *Information & Management*, *51*, 13-26.

Zimmerman, T., Premraj, R., Bettenburg, N., Just, S., Schröter, A., & Weiss, C. (2010). What makes a good bug report? *IEEE Transactions on Software Engineering*, *36*(5), 618-643.

About the Authors

Michael J. Lee is a PhD candidate of Information Systems in the David Eccles School of Business at the University of Utah. Michael has over 20 years of experience in software management and data analytics in industry, in both the public and private sectors. Building on this foundation, his primary research interests are in software management, healthcare information systems and social media analytics. Michael is the author of seven practitioner books related to software and data management, specifically pertaining to the Microsoft development technology stack.

Marcus A. Rothenberger is a Professor of MIS in the Department of Management, Entrepreneurship, and Technology of the Lee Business School at the University of Nevada Las Vegas. He holds a Ph.D. in Information Systems from Arizona State University. His work includes theory testing, theory development, and design science research in the areas of software process improvement, software reusability, performance measurement, and the adoption of Enterprise Resource Planning systems. His work has appeared in major academic journals, such as the *Journal of Management Information Systems*, the *Decision Sciences Journal*, *IEEE Transactions on Software Engineering*, *IEEE Transactions on Engineering Management*, *Communications of the ACM*, *Information & Management*, and the *Journal of Information Technology Theory and Application*. He is co-editor-in-chief of *JITTA*.

Ken Peffers is Professor of MIS, Lee School of Business, University of Nevada Las Vegas. He holds a Ph.D. from Purdue University. His research followed three broad themes: evaluating IT investments, design research, and service system requirements. He was the founding Editor-in-Chief and publisher of the *Journal of Information Technology Theory & Application,* now an AIS publication. His 2007-8 paper in *JMIS*, "A Design Science Research Methodology for Information Systems Research", has been cited more than any other paper in information systems research published since its print date (March 2008).

JITTA

JOURNAL OF INFORMATION TECHNOLOGY THEORY AND APPLICATION

JITTA is a Publication of the Association for Information Systems
ISSN: 1532-3416